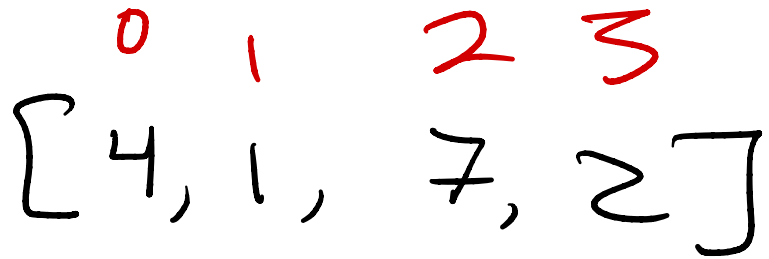
Lecture 15

Persistent arrays / Sequences

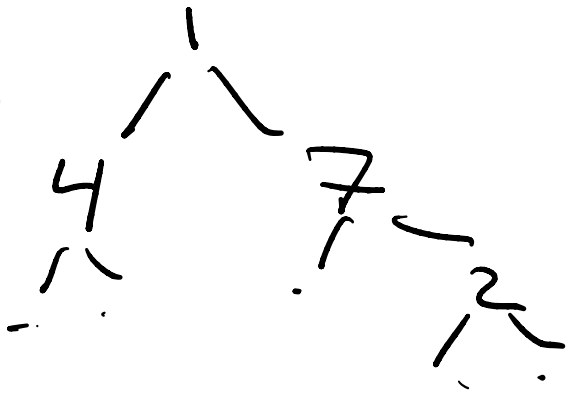
Ordered collection of elements

Implementation → differ in time, work, or span

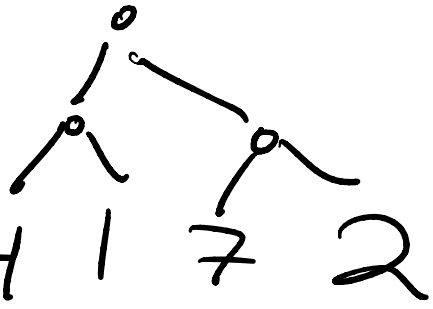
1. Lists



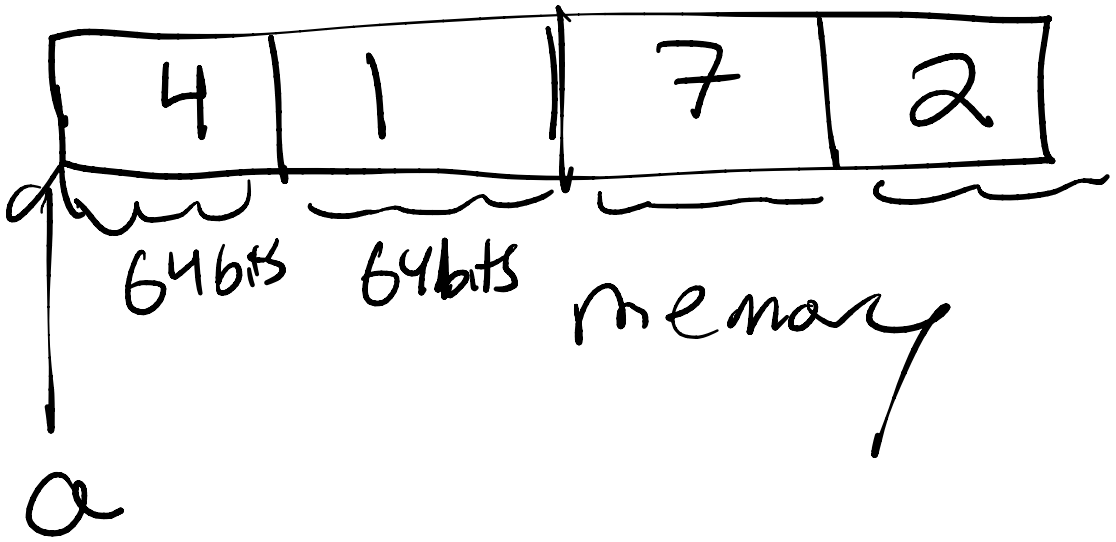
2. Trees (internal)



3. Trees (leaf)



4 arrays



Basic operations on arrays

1. get element at position i

O(1) work

a[i] nth(i, a)

2. set element at position i

O(1) work

a[i] = e set(a, i, e)

Can do these operations for other ordered collections

n = length

```
(* assume 0 ≤ i < length l *)
fun nth(l : 'a list, i : int) : 'a =
    case (l, i) of
        (x::xs, 0) => x
      | (x::xs, _) => nth(xs, i-1)
```

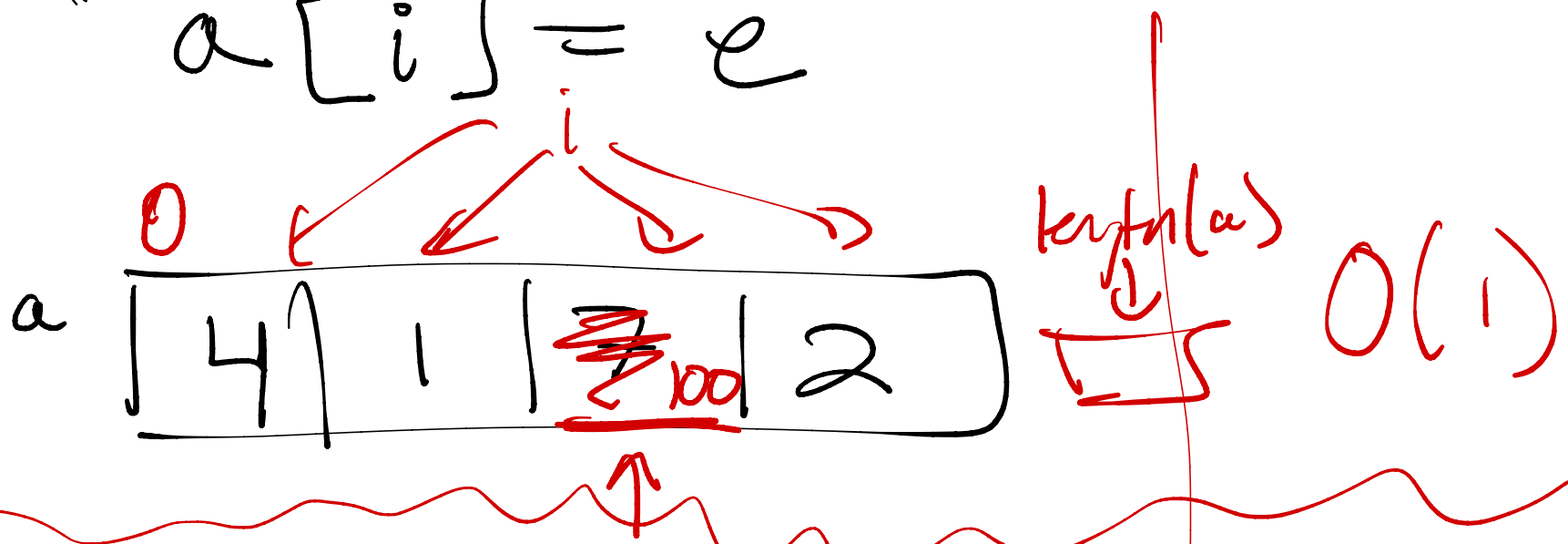
work $O(n)$
span $= O(n)$
(really $O(i)$)

---

eg. nth([4,1,7,2], 2) = nth([1,7,2], 1)
= 7

(0 1 2 3)

"a[i] = e"



a[2] = 100 set(a, 2, 100)

(* assume 0 ≤ i < length(l) *)

```
fun set (l : 'a list, i : int, new : 'a) : 'a list =
  case (l, i) of
    (x::xs, 0) => new :: xs
  | (x::xs, _) => x :: set(xs, i-1, new)
```

w/s O(n) (n = length(l))

e.g. set([4,1,7,2], 2, 100) = 4 :: set([1,7,2], 1, 100)

	list	array
nth	O(n)	O(1)
set	O(n)	O(1)

Persistent Data

set([4, 1, 7, 2], 2, 100)

3, 4, 5, 6, 12, ----]

VS

⟼ [4, 1, 100,]

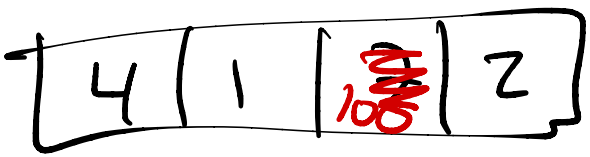
O(n) time

O(n) space

Ephemeral data

Downside: action at a distance

O(1) set work space

a 

a[2] = 100

original array no longer exists

Persistent arrays: ordered collection of elements

→ each operation makes a new/copied array

type 'a seq [read: sequence]

nth : int * 'a seq → 'a

a[i]=e set : 'a seq * int * 'a → 'a seq

length : 'a seq → int

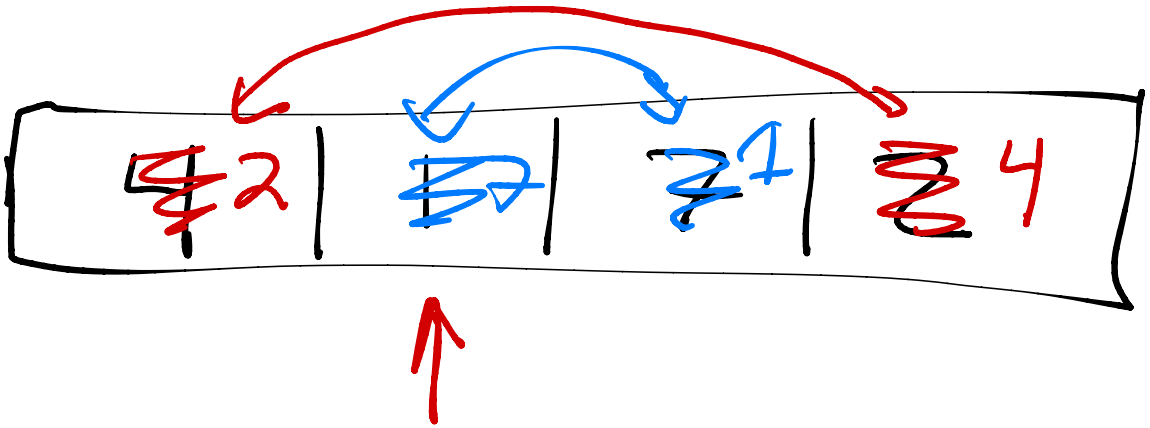
a i e

new copy

O(n) work

span

Translitwate C code



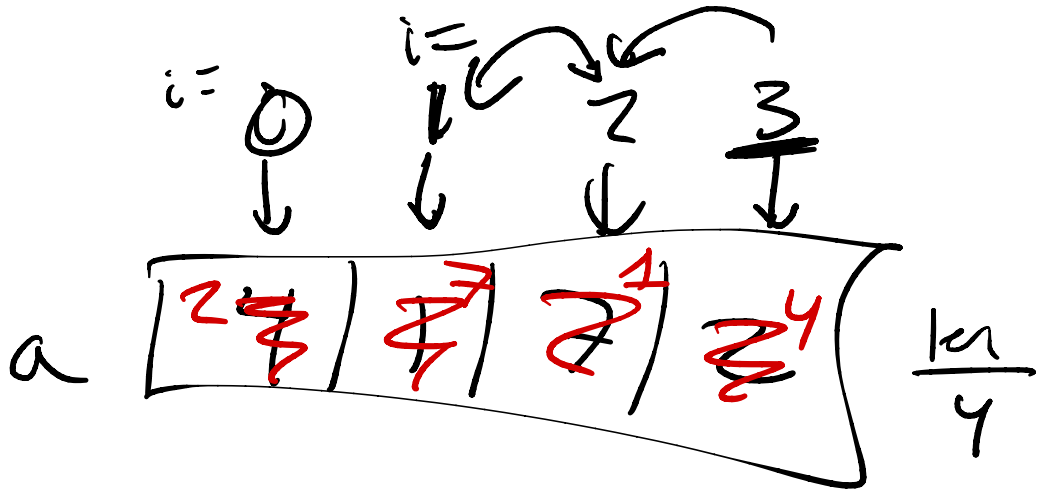
reverse

2	7	1	4

```
for (i=0; i<= length(a) div 2 ; i++) {
```

```
first = a[i]
last = a[length(a)-1-i]
a[ i ] = last
a[length(a)-1-i] = first
```

```
}
```



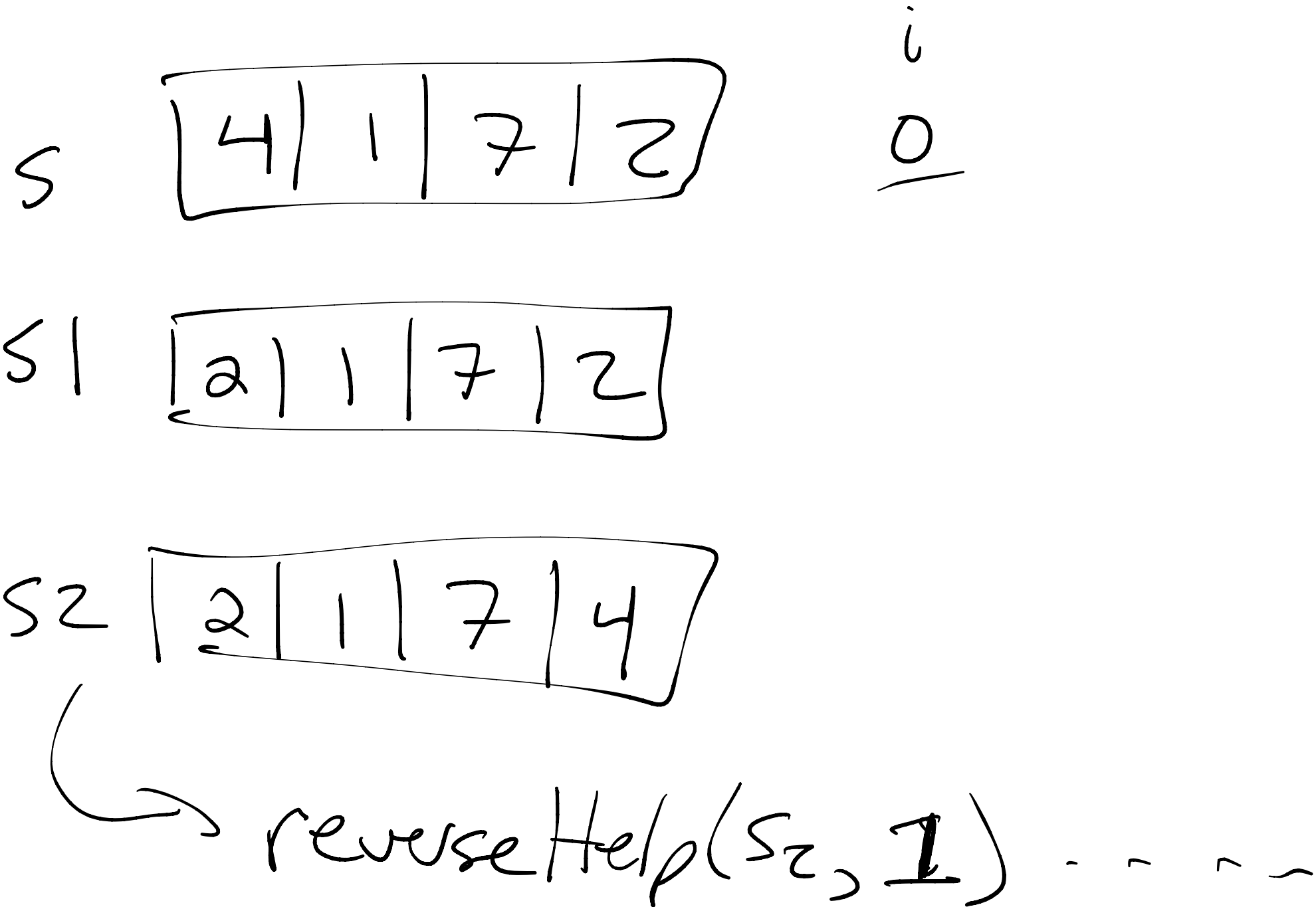
```
fun reverse(s: 'a seq): 'a seq =
    reverseHelp(s, 0)

fun reverseHelp(s: 'a seq, i: int): 'a seq =
  case i = length(s) div 2 of
    true => s
  | false =>
      let val first = nth(s, i)
          val last = nth(s, length(s)-1-i)
          val s1 = set(s, i, last)
          val s2 = set(s1, length(s)-1-i, first)
      in reverseHelp(s2, i+1)
      end
```

for(i=0; i<=length(s)/2; i++)

copy1

copy2


i
s
4 1 7 2
0
s1
2 1 7 2
s2
2 1 7 4
reverseHelp(s2, 1) - - - -

sets at each step $O(n)$ time

↳ and also use $O(n)$ new memory to make copies

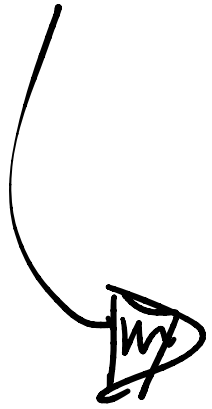
↳ $O(n)$ times loop

$O(n^2)$ work

$O(n^2)$ new memory

Focus on "bulk operations" (not set a[i] = e)

on persistent arrays



represent using higher-order functions

type 'a seq

nth : int * 'a seq → 'a

a[i]=e

set : 'a seq * int * 'a → 'a seq

length : 'a seq → int

increasing : int → int seq

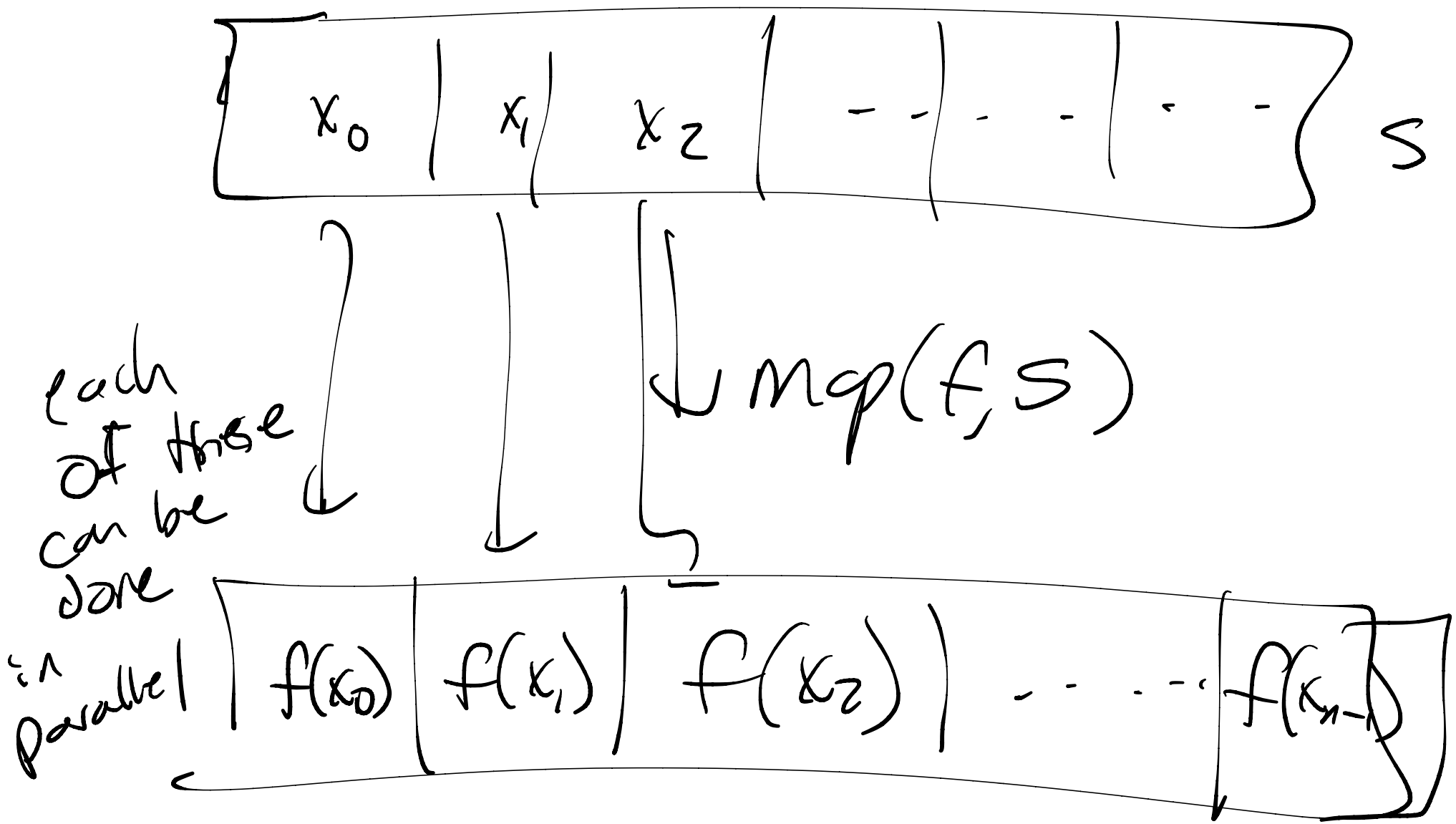
increasing(n) =

| 0 | 1 | 2 | 3 | (...) | n-1 |

map : ('a → 'b) * 'a seq → 'b seq

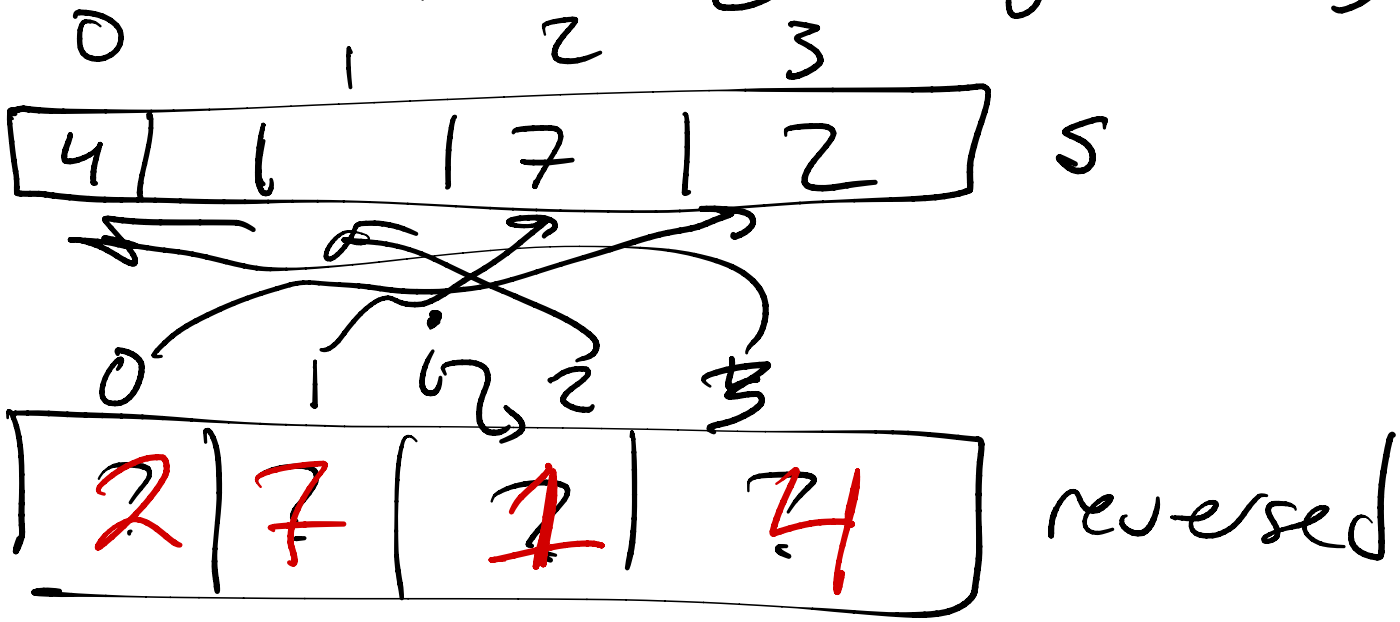
reduce : ('a * 'a → 'a) * 'a * 'a seq → 'a

filter : ('a → bool) * 'a seq → 'a seq


x_0 x_1 x_2 - - - - - - - - s
map(f, s)
each of these can be done in parallel
f(x0) f(x1) f(x2) ... f(xn-1)

new copied array

```
fun reverse(s: 'a seq): 'a seq =
  map (fn i =>
         nth(s, length(s)-1-i))
      (increasing(length s))
```



Think of these array ops as

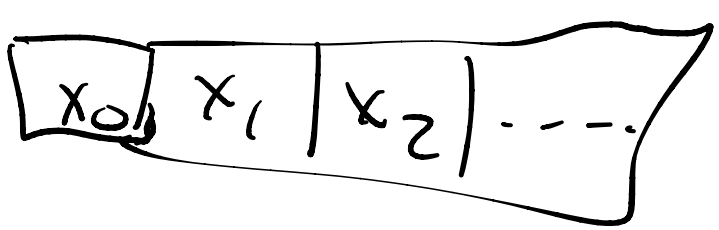
built-in +

abstractly specify behavior

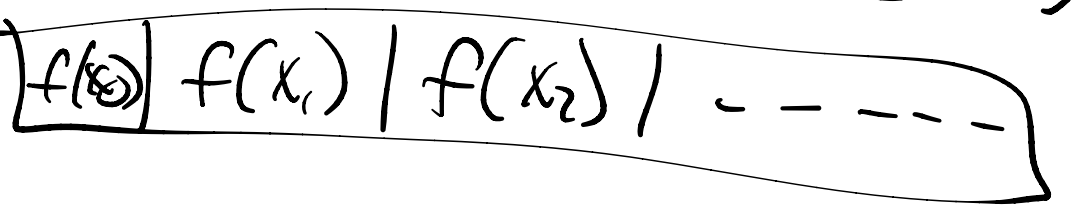
and the time

map : ('a → 'b) * 'a seq → 'b seq

Behavior

map(f, 

)

= 

Work

If f is O(1)
O(n)

If not
$W_f(x_0) + W_f(x_1) + \cdots + W_f(x_{n-1})$

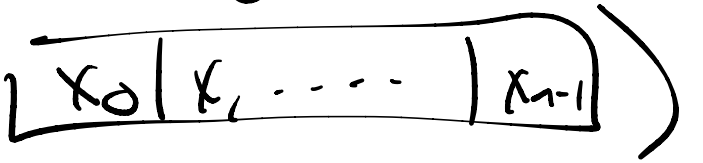
Span

"n-way parallelism"
O(1)

max of
$S_f(x_0), S_f(x_1), \ldots$

nth : int * 'a seq → 'a

nth (i, s) = x_i



if $0 \leq i <$ length(s)

or raise Range o.w.

Time work $O(1)$

span $O(1)$

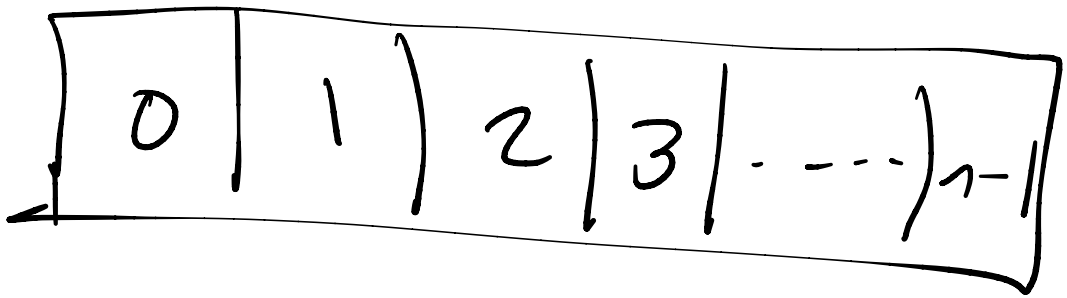
length : 'a seq → int

length ($\langle x_0 | x_1 | \cdots | x_{n-1} \rangle$) = n

Work $O(1)$

Span $O(1)$

increasing : int → int seq

increasing n = 

work $O(n)$

span $O(1)$

filter

Behavior filter(p, $\langle x_0, x_1, \ldots, x_{n-1} \rangle$)

those x_i such that $p(x_i)$ = true

Work If p constant $O(n)$

Span $O(\log n)$